# CMSC 330:
# Organization of Programming Languages

## OCaml Basics

Spring 2026

# OCaml Compiler

- OCaml programs can be compiled using
-
  - ocamlc: produces bytecode
    - Faster to compile
    - Slower to run
  - ocamlopt: produces native code
    - Faster, but not platform-independent (or as easily debugged)

# OCaml Compiler

- Compiling and running the following small program:

hello.ml:

```
(* A small OCaml program *)
print_string "Hello world!\n";;
```

```
% ocamlc hello.ml
% ./a.out
Hello world!
```

OCaml interpreter

```
% ocaml hello.ml
Hello world!
```

# OCaml Compiler: Multiple Files

main.ml:

```
let main () =
  print_int (Util.add 10 20);
  print_string "\n"

let () = main ()
```

util.ml:

```
let add x y = x+y
```

- Compile both together (produces `a.out`)

  ```
  ocamlc util.ml main.ml
  ```

- Or compile separately

  ```
  ocamlc -c util.ml
  ocamlc util.cmo main.ml
  ```

- To execute

  ```
  ./a.out
  ```

# OCaml Top-level

- The *top-level* is a read-eval-print loop (REPL) for OCaml
- Start the top-level via the **`ocaml`** command

```
#ocaml
    % OCaml version 5.4.0
    print_string "Hello world!\n";;
    Hello world!
```

- **`utop`** is an alternative top-level; improves on **`ocaml`**
- To exit the top-level, type ^D (Control D) or call the exit 0

```
exit 0;;
```

# Loading Code Files into the Top-level

File `hello.ml` :

```
print_string "Hello world!\n";;
```

- Load a file into top-level

  `#use "filename.ml"`

- Example:  #use processes a file a line at a time

```
# #use "hello.ml";;
Hello world!
- : unit = ()
#
```

# OPAM: OCaml Package Manager

- **`opam`** is the package manager for OCaml
  - Manages libraries and different compiler installations

- You should install the following packages with **`opam`**
  - **`utop`**, a top-level interface
  - **`dune`**, a build system for larger projects

# Project Builds with **`dune`**

- Use **`dune`** to compile projects---automatically finds dependencies, invokes compiler and linker
- Define a **`dune`** file, similar to a **`Makefile`**:

```
dune init project HelloWorld
cd HelloWorld
dune build
_build/default/main.exe
```

# Dune commands

- If defined, run a project's test suite:

```
dune runtest
```

- Load the modules defined in src/ into the **`utop`** top-level interface:

```
dune utop src
```

# Functional Programming with OCaml

A functional language:

- defines computations as **mathematical functions**
- *discourages* use of **mutable state,** the information maintained by a computation

# Functional vs. Imperative

## Functional languages

- *Higher* level of abstraction: *What* to compute, not *how*
- *Immutable* state: easier to reason about (meaning)
- *Easier* to develop robust software

## Imperative languages

- *Lower* level of abstraction: *How* to compute, not *what*
- *Mutable* state: harder to reason about (behavior)
- *Harder* to develop robust software

# Functional vs. Imperative

```
sum = 0
for x in [1,2,3,4]:
    sum = sum + x
print(sum)
```

```
List.fold_left (+) 0 [1;2;3;4]
```

Imperative: Emphasizes *control flow*.

Functional: Emphasizes *declarative style*.

# Functional vs. Imperative

Functional (expression-based, no mutation)

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)
```

- No variables are updated.
- Defined as a mathematical function.
- Describes what factorial is: $n! = n \times (n-1)!$

Imperative (step-by-step, with mutable state)

```
int fact(int n){
    int result = 1;
    int i = 1;
    while i <= n{
     result = result * i;
     i = i + 1;
    }
    return result;
}
```

- Uses variables (result, i) that change over time.
- Uses a loop to control execution.
- Describes how to compute the factorial step by step.

# ML-style (Functional) Languages

- ML (Meta Language)
  - Univ. of Edinburgh, 1973
  - Part of a theorem proving system LCF
- Standard ML
  - Bell Labs and Princeton, 1990; Yale, AT&T, U. Chicago
- OCaml (Objective CAML)
  - INRIA, 1996
    - French Nat'l Institute for Research in Computer Science
  - O is for "objective", meaning objects (which we'll ignore)
- Haskell (1998): *lazy* functional programming
- Scala (2004): functional and OO programming

# Key Features of ML

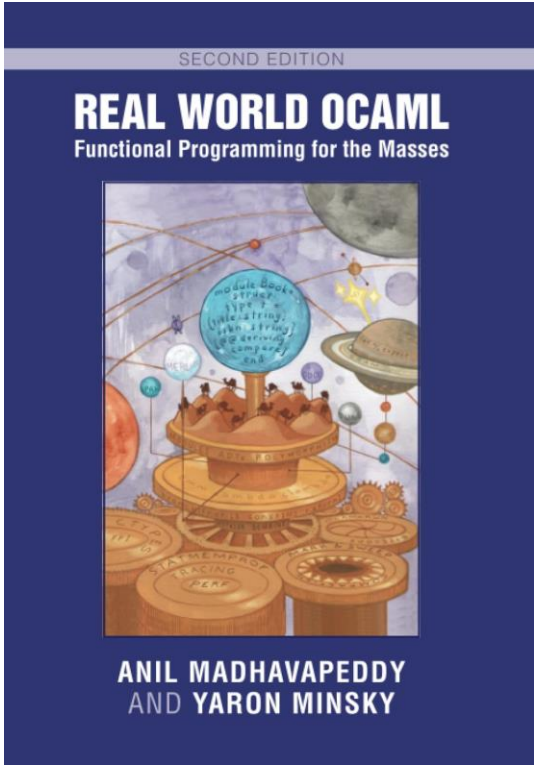
- First-class functions
  - Functions can be parameters to other functions ("higher order") and return values, and stored as data
- Favor immutability ("assign once")
- Data types and pattern matching
  - Convenient for certain kinds of data structures
- Type inference
  - No need to write types in the source language
    - But the language is statically typed
  - Supports parametric polymorphism
    - *Generics* in Java, *templates* in C++
- Exceptions and garbage collection

# Why study functional programming?

**Many modern languages are influenced by Functional languages:**

- Garbage collection
  - LISP [1958], Java [1995], Python 2 [2000], Go [2007]
- Parametric polymorphism (generics)
  - ML [1973], SML [1990], Java 5 [2004], Rust [2010]
- Higher-order functions
  - LISP [1958], Haskell [1998], Python 2 [2000], Swift [2014]
- Type inference
  - ML [1973], C++11 [2011], Java 7 [2011], Rust [2010]
- Pattern matching
  - SML [1990], Scala [2002], Rust [2010], Java 16 [2021]

# Recommended Textbook




Free online:
https://dev.realworldocaml.org/

# Similar Courses

- CS3110 (Cornell)
- CSE341 (Washington)
- 601.426 (Johns Hopkins)
- COS326 (Princeton)
- CS152 (Harvard)
- CS421 (UIUC)

# Other Resources

- Cornell cs3110 book is another course which uses OCaml; it is more focused on programming and less on PL theory than this class is.
- ocaml.org is the home of OCaml for finding downloads, documentation, etc. The tutorials are also very good and there is a page of books.
- OCaml from the very beginning is a free online book.

# OCaml Coding Guidelines

- We will not grade on style, but style is important
- Recommended coding guidelines:

- https://ocaml.org/learn/tutorials/guidelines.html

# Lecture Presentation Style

- Our focus: semantics and idioms for OCaml
  - *Semantics* is what the language does
  - *Idioms* are ways to use the language well

- We will also cover some useful libraries

- Syntax is what you type, not what you mean
  - In one lang: Different syntax for similar concepts
  - Across langs: Same syntax for different concepts
  - Syntax can be a source of fierce disagreement among language designers!

# Expressions

- Expressions are our primary building block
  - Akin to *statements* in imperative languages
- Every kind of expression has
  - Syntax
    - We use metavariable *e* to designate an arbitrary expression
  - Semantics
    - Type checking rules (static semantics): produce a type or fail with an error message
    - Evaluation rules (dynamic semantics): produce a value
      - (or an exception or infinite loop)
      - Used *only* on expressions that type-check

# Values

- A value is an expression that is final
  - **34** is a value, **true** is a value
  - **34+17** is an *expression*, but *not* a value

- Evaluating an expression means running it until it's a value
  - `34+17` *evaluates* to `51`

$$e \rightarrow v$$

# Types

- Types classify expressions
  - The set of values an expression could evaluate to
  - We use metavariable `t` to designate an arbitrary type
- Expression `e` has type `t` if `e` will (always) evaluate to a value of type `t`
- Write `e : t` to say `e` has type `t`.
- Determining that `e` has type `t` is called type checking

1: int  true:bool

3+4: int  “hello”:string

# If Expressions

- Syntax: **`if e1 then e2 else e3`**
- Type checking

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e3 : \tau}{\Gamma \vdash \texttt{if e1 then e2 else e3} : \tau}$$

  - It reads: **`if e1 then e2 else e3`** has type **`t`** if
    - **`e1`** has type **`bool`**
    - Both **`e2`** and **`e3`** have type **`t`** (for some **`t`**)

# If Expressions: Type Checking and Evaluation

```
if 7 > 42 then "hello" else “goodbye";;

- : string = "goodbye”
```

# If Expressions: Type Checking and Evaluation

```
if false then 3 else 3.0;;
```

```
Error: This expression has type float but
an expression was expected of type int
```

# If Expressions: Another Example

```
If 10 < 20 then print_int 10;;
```

Same as

```
If 10 < 20 then print_int 10 else ( );;
```

# Functions

- OCaml functions are like mathematical functions
  - Compute a result from provided arguments

```
let next x = x + 1;;

next 10;;

next : int -> int = <fun>
- : int = 11
```

# Recursive Functions

```
(* requires n>= 0
   returns: n! *)
let rec fact n =
   if n = 0 then
     1
   else
     n * fact (n-1)
```

# Function Types

- In OCaml, **`->`** is the function type constructor
  - Type ***`t1 -> t`*** is a function with argument or *domain* type ***`t1`*** and return or *range* type ***`t`***
  - Type ***`t1 -> t2 -> t`*** is a function that takes *two* inputs, of types ***`t1`*** and ***`t2`***, and returns a value of type ***`t`***. Etc.
- Examples
  - **`not`**
  - **`int_of_float`**
  - **`+`**

```
(* type bool -> bool *)

(* type float -> int *)

(* type int -> int -> int *)
```

# Type Inference

- A declared variable need not be annotated with its type
  - The type can be inferred
  - Type inference happens *as a part of type checking*
  - Determines a type that satisfies code's constraints

- What is the type of:

```
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-1)
```

# Type Checking: Calling Functions

- Syntax **`f e1 … en`**
- Type checking

$$\frac{\Gamma \vdash f : t_1 \to t_2 \ldots \to t_n \to u, \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \ldots \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \texttt{f e1 e2 .. en} : u}$$

It read: if **`f : t1 -> … -> tn -> u ,`** and **`e1 : t1 , … , en : tn`** then **`f e1 … en : u`**

- Example:
  - **`not true : bool`**
  - since **`not : bool -> bool`** and **`true : bool`**

# Calling Functions: Evaluation

```
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-1)
```

```
fact 2
if 2=0 then 1 else 2*fact(2-1)
2 * fact 1
2 * (if 1=0 then 1 else 1*fact(1-1))
2 * 1 * fact 0
2 * 1 * (if 0=0 then 1 else
0*fact(0-1))
2 * 1 * 1
2
```

# Function Type Checking: More Examples

```
let double x = x * 2;;

double:  int -> int
```

double is the name of the function. Not the data type in C or Java.

# Function Type Checking: More Examples

```
let fn x = (int_of_float x) * 3;;

Type of fn: float -> int
```

# Mutually Recursrive Functions

```
let rec odd n =
    if n == 0 then false
    else even(n-1)
 and
    even n =
      if n == 0 then true
      else odd(n-1);;
```

# Polymorphic Types

```
let eq x y = (x = y);;
```

```
eq:'a -> 'a -> bool
```

# Type annotations

we can provide type annotations manually.

```
let (x : int) = 3;; (* binds x to 3 *)

let add (x:int) (y:int):int = x + y;;

let id x = x;; (*identity function *)

let id (x:int) = x;;
```